4 Authenticated encryption

This chapter covers

- Symmetric encryption vs. authenticated encryption
- Popular authenticated encryption algorithms
- Other types of symmetric encryption

Confidentiality is about hiding data from unwanted eyes and encryption is the way to achieve this. Encryption is what the science of cryptography was initially invented for; it's what preoccupied most of the early cryptographers. They would ask themselves, "How can we prevent observers from understanding our conversations?" While the science and its advances first bloomed behind closed doors, benefiting the governments and their militaries only, it is now opened throughout the world. Today, encryption is used everywhere to add privacy and security in the different aspects of our modern lives. In this chapter, we'll find out what encryption really is, what types of problems it solves, and how today's applications make heavy use of this cryptographic primitive.

NOTE For this chapter, you'll need to have read chapter 3 on message authentication codes.

4.1 What's a cipher?

It's like when you use slang to talk to your siblings about what you'll do after school so your mom doesn't know what you're up to.

—Natanael L. (2020, https://twitter.com/Natanael_L)

Let's imagine that our two characters, Alice and Bob, want to exchange some messages privately. In practice, they have many mediums at their disposal (the mail, phones, the internet, and so on), and each of these mediums are by default insecure. The mailman could open their letters; the telecommunication operators can spy on their calls and text messages; internet service providers or any servers on the network that are in between Alice and Bob can access the content of the packets being exchanged.

Without further ado, let's introduce Alice and Bob's savior: the *encryption algorithm* (also called a *cipher*). For now, let's picture this new algorithm as a black box that Alice can use to encrypt her messages to Bob. By *encrypting* a message, Alice transforms it into something that looks random. The encryption algorithm for this takes

- *A secret key*—It is crucial that this element is unpredictable, random, and well protected because the security of the encryption algorithm relies directly on the secrecy of the key. I will talk more about this in chapter 8 on secrets and randomness.
- *Some plaintext*—This is what you want to encrypt. It can be some text, an image, a video, or anything that can be translated into bits.

This encryption process produces a *ciphertext*, which is the encrypted content. Alice can safely use one of the mediums listed previously to send that ciphertext to Bob. The ciphertext will look random to anyone who does not know the secret key, and no information about the content of the message (the plaintext) will be leaked. Once Bob receives this ciphertext, he can use a *decryption algorithm* to revert the ciphertext into the original plaintext. Decryption takes

- *A secret key*—This is the same secret key that Alice used to create the ciphertext. Because the same key is used for both algorithms, we sometimes call the key a *symmetric key*. This is also why we also sometimes specify that we are using *symmetric encryption* and not just *encryption*.
- *Some ciphertext*—This is the encrypted message Bob receives from Alice.

The process then reveals the original plaintext. Figure 4.1 illustrates this flow.

Encryption allows Alice to transform her message into something that looks random and that can be safely transmitted to Bob. Decryption allows Bob to revert the encrypted message back to the original message. This new cryptographic primitive provides confidentiality (or secrecy or privacy) to their messages.

NOTE How do Alice and Bob agree to use the same symmetric key? For now, we'll assume that one of them had access to an algorithm that generates

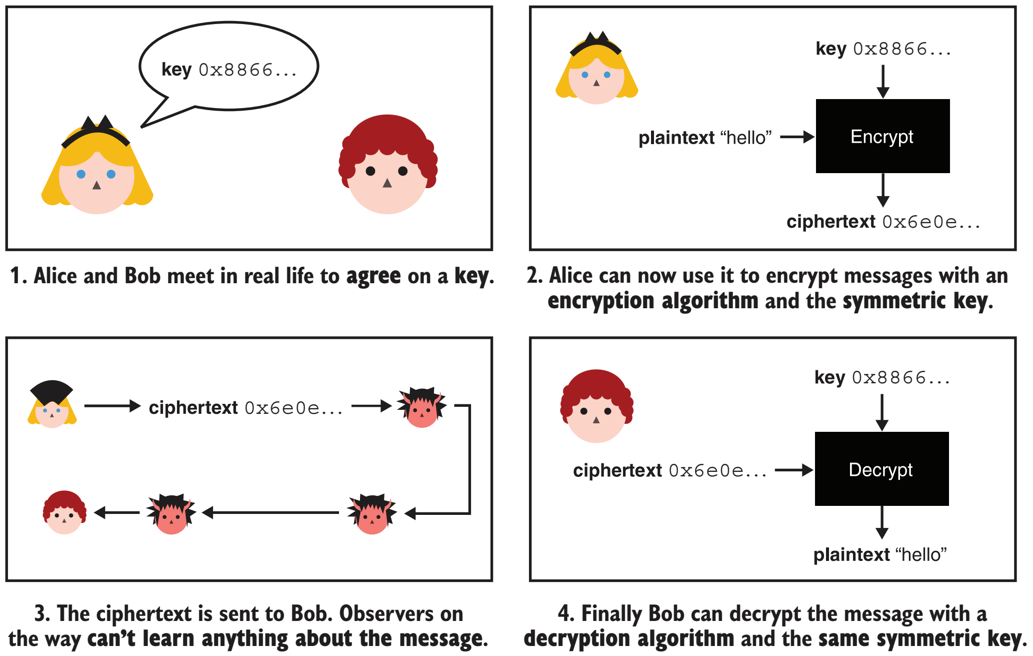

key 0x8866...
1. Alice and Bob meet in real life to agree on a key.
key 0x8866...
plaintext "hello"
Encrypt
ciphertext 0x6e0e...
2. Alice can now use it to encrypt messages with an encryption algorithm and the symmetric key.
ciphertext 0x6e0e...
3. The ciphertext is sent to Bob. Observers on the way can't learn anything about the message.
key 0x8866...
ciphertext 0x6e0e...
Decrypt
plaintext "hello"
4. Finally Bob can decrypt the message with a decryption algorithm and the same symmetric key.

Figure 4.1 Alice (top right) encrypts the plaintext *hello* with the key `0x8866...` (an abbreviated hexadecimal). Then Alice sends the ciphertext to Bob. Bob (bottom right) decrypts the received ciphertext by using the same key and a decryption algorithm.

unpredictable keys, and that they met in person to exchange the key. In practice, how to bootstrap such protocols with shared secrets is often one of the great challenges companies need to solve. In this book, you will see many different solutions to this problem.

Notice that I have yet to introduce what the title of this chapter, "Authenticated encryption," refers to. I've only talked about encryption alone so far. While encryption alone is not secure (more about that later), I have to explain how it works before I can introduce the authenticated encryption primitive. So bear with me as I first go over the main standard for encryption: the *Advanced Encryption Standard* (AES).

4.2 The Advanced Encryption Standard (AES) block cipher

In 1997, NIST started an open competition for an *Advanced Encryption Standard* (AES), aimed at replacing the Data Encryption Standard (DES) algorithm, their previous standard for encryption that was starting to show signs of age. The competition lasted three years, during which time, 15 different designs were submitted by teams of cryptographers from different countries. At the end of the competition, only one submission, Rijndael, by Vincent Rijmen and Joan Daemen was nominated as the winner.

In 2001, NIST released AES as part of the FIPS (Federal Information Processing Standards) 197 publication. AES, the algorithm described in the FIPS standard, is still the main cipher used today. In this section, I explain how AES works.

4.2.1 How much security does AES provide?

AES offers three different versions: AES-128 takes a key of 128 bits (16 bytes), AES-192 takes a key of 192 bits (24 bytes), and AES-256 takes a key of 256 bits (32 bytes). The length of the key dictates the level of security—*the bigger, the stronger.* Nonetheless, most applications make use of AES-128 as it provides enough security (128 bits of security).

The term *bit security* is commonly used to indicate the security of cryptographic algorithms. For example, AES-128 specifies that the best attack we know of would take around 2^{128} operations. This number is gigantic, and it is the security level that most applications aim for.

Bit security is an upper bound

The fact that a 128-bit key provides 128 bits of security is specific to AES; it is not a golden rule. A 128-bit key used in some other algorithm could theoretically provide less than 128-bit security. While a 128-bit key can provide less than 128-bit security, it will never provide more (there's always the brute force attack). Trying all the possible keys would take at most 2^{128} operations, reducing the security to 128 bits at least.

How big is 2^{128}? Notice that the amount between two powers of 2 is doubled. For example 2^3 is twice as much as 2^2. If 2^{100} operations are pretty much impossible to reach, imagine achieving double that (2^{101}). To reach 2^{128}, you have doubled your initial amount 128 times! In plain English, 2^{128} is 340 undecillion 282 decillion 366 nonillion 920 octillion 938 septillion 463 sextillion 463 quintillion 374 quadrillion 607 trillion 431 billion 768 million 211 thousand 456. It is quite hard to imagine how big that number is, but you can assume that we will never be able to reach such a number in practice. We also didn't account for the amount of space required for any large and complex attack to work, which is equally as enormous in practice.

It is foreseeable that AES-128 will remain secure for a long time. That is unless advances in cryptanalysis find a yet undiscovered vulnerability that would reduce the number of operations needed to attack the algorithm.

4.2.2 The interface of AES

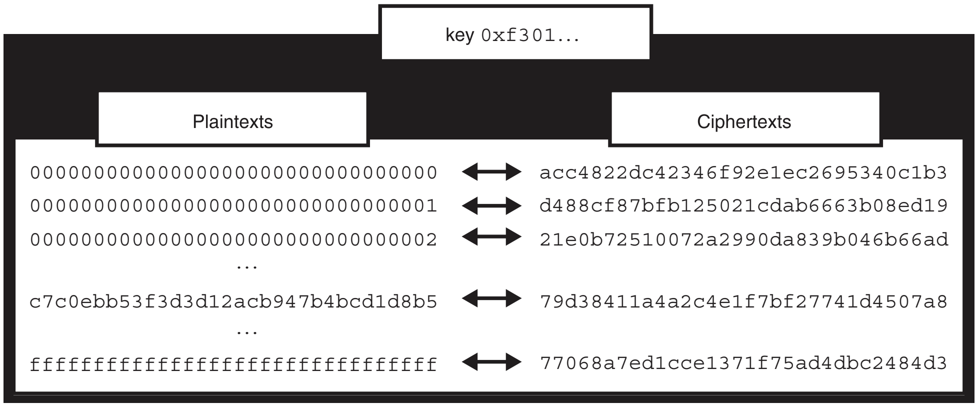
Looking at the interface of AES for encryption, we see the following:

- The algorithm takes a variable-length key as discussed previously.
- It also takes a plaintext of exactly 128 bits.
- It outputs a ciphertext of exactly 128 bits.

Because AES encrypts a fixed-size plaintext, we call it a *block cipher*. Some other ciphers can encrypt arbitrarily length plaintexts as you will see later in this chapter.

The decryption operation is exactly the reverse of this: it takes the same key, a ciphertext of 128 bits, and returns the original 128-bit plaintext. Effectively, decryption reverts the encryption. This is possible because the encryption and decryption operations are *deterministic*; they produce the same results no matter how many times you call them.

In technical terms, a block cipher with a key is a *permutation*: it maps all the possible plaintexts to all the possible ciphertexts (see the example in figure 4.2). Changing the key changes that mapping. A permutation is also reversible. From a ciphertext, you have a map back to its corresponding plaintext (otherwise, decryption wouldn't work).



```
Plaintexts                                  Ciphertexts
00000000000000000000000000000000   <-->   acc4822dc42346f92e1ec2695340c1b3
00000000000000000000000000000001   <-->   d488cf87bfb125021cdab6663b08ed19
00000000000000000000000000000002   <-->   21e0b72510072a2990da839b046b66ad
               ...
c7c0ebb53f3d3d12acb947b4bcd1d8b5   <-->   79d38411a4a2c4e1f7bf27741d4507a8
               ...
ffffffffffffffffffffffffffffffff   <-->   77068a7ed1cce1371f75ad4dbc2484d3
```

Figure 4.2 A cipher with a key can be seen as a permutation: it maps all the possible plaintexts to all the possible ciphertexts.

Of course, we do not have the room to list all the possible plaintexts and their associated ciphertexts. That would be 2^{128} mappings for a 128-bit block cipher. Instead, we design constructions like AES, which behave like permutations and are randomized by a key. We say that they are *pseudorandom permutations* (PRPs).

4.2.3 The internals of AES

Let's dig a bit deeper into the guts of AES to see what's inside. Note that AES sees the *state* of the plaintext during the encryption process as a 4-by-4 matrix of bytes (as you can see in figure 4.3).

This doesn't really matter in practice, but this is how AES is defined. Under the hood, AES works like many similar symmetric cryptographic primitives called *block ciphers*, which are ciphers that encrypt fixed-sized blocks. AES also has a *round function*

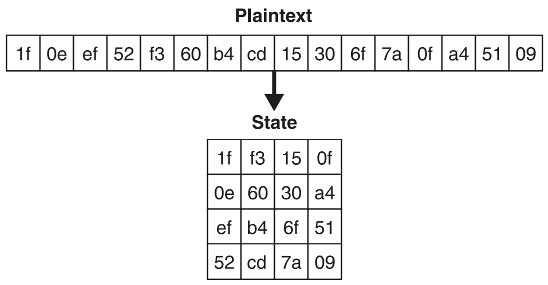


Figure 4.3 When entering the AES algorithm, a plaintext of 16 bytes gets transformed into a 4-by-4 matrix. This state is then encrypted and finally transformed into a 16-byte ciphertext.

that it iterates several times, starting on the original input (the plaintext). I illustrate this in figure 4.4.

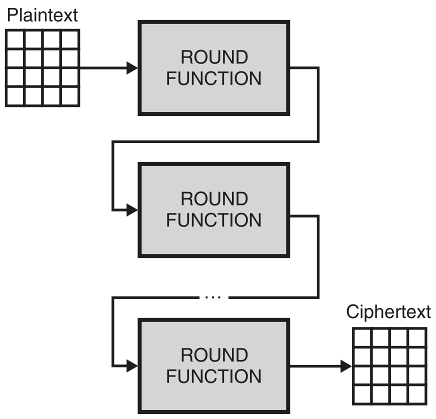


Figure 4.4 AES iterates a round function over a state in order to encrypt it. The round function takes several arguments including a secret key. (These are missing from the diagram for simplicity.)

Each call to the round function transforms the state further, eventually producing the ciphertext. Each round uses a different *round key*, which is derived from the main symmetric key (during what is called a *key schedule*). This allows the slightest change in the bits of the symmetric key to give a completely different encryption (a principle called *diffusion*).

The round function consists of multiple operations that mix and transform the bytes of the state. The round function of AES specifically makes use of four different subfunctions. While we will shy away from explaining exactly how the subfunctions work (you can find this information in any book about AES), they are named `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The first three are easily reversible (you can find the input from the output of the operation), but the last one is not. It performs an exclusive OR (XOR) with the round key and the state and, thus, needs the knowledge of the round key to be reversed. I illustrate what goes into a round in figure 4.5.

The number of iterations of the round function in AES, which are usually practical on a reduced number of rounds, was chosen to thwart cryptanalysis. For example,

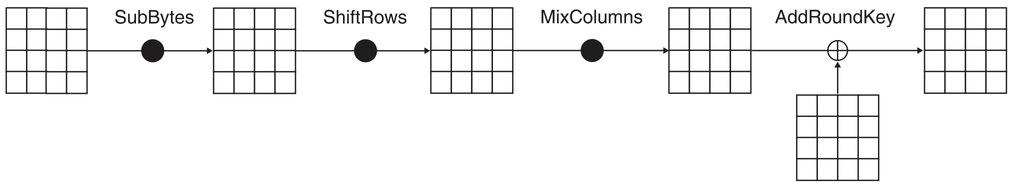


Figure 4.5 A typical round of AES. (The first and last rounds omit some operations.) Four different functions transform the state. Each function is reversible as decryption wouldn't work otherwise. The addition sign inside a circle (⊕) is the symbol for the XOR operation.

extremely efficient *total breaks* (attacks that recover the key) exist on three round variants of AES-128. By iterating many times, the cipher transforms plaintext into something that looks nothing like the original plaintext. The slightest change in the plaintext also returns a completely different ciphertext. This principle is called the *avalanche effect.*

NOTE Real-world cryptographic algorithms are typically compared by the security, size, and speed they provide. We already talked about the security and size of AES; its security depends on the key size, and it can encrypt 128-bit blocks of data at a time. Speedwise, many CPU vendors have implemented AES in hardware. For example, AES New Instructions (AES-NI) is a set of instructions available in Intel and AMD CPUs, which can be used to efficiently implement encryption and decryption for AES. These special instructions make AES extremely fast in practice.

One question that you might still have is how do I encrypt more or less than 128 bits with AES? I'll answer this next.

4.3 The encrypted penguin and the CBC mode of operation

Now that we have introduced the AES block cipher and explained a bit about its internals, let's see how to use it in practice. The problem with a block cipher is that it can only encrypt a block by itself. To encrypt something that is not exactly 128 bits, we must use a *padding* as well as a *mode of operation.* So let's see what these two concepts are about.

Imagine that you want to encrypt a long message. Naively, you could divide the message into blocks of 16 bytes (the block size of AES). Then if the last block of plaintext is smaller than 16 bytes, you could append some more bytes at the end until the plaintext becomes 16 bytes long. This is what padding is about!

There are several ways to specify how to choose these *padding bytes,* but the most important aspect of padding is that it must be reversible. Once we decrypt ciphertext, we should be able to remove the padding to retrieve the original unpadded message. Simply adding random bytes, for example, wouldn't work because you wouldn't be able to discern if the random bytes were part of the original message or not.

The most popular padding mechanism is often referred to as *PKCS#7 padding*, which first appeared in the PKCS#7 standard published by RSA (a company) at the end of the 1990s. PKCS#7 padding specifies one rule: the value of each padding byte must be set to the length of the required padding. What if the plaintext is already 16 bytes? Then we add a full block of padding set to the value 16. I illustrate this visually in figure 4.6. To remove the padding, you can easily check the value of the last byte of plaintext and interpret it as the length of padding to remove.

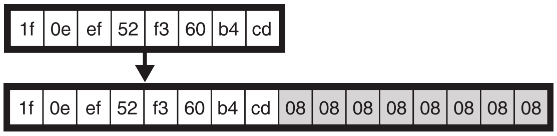


Figure 4.6 If the plaintext is not a multiple of the block size, it is padded with the length needed to reach a multiple of the block size. In the figure, the plaintext is 8 bytes, so we use 8 more bytes (containing the value 8) to pad the plaintext up to the 16 bytes required for AES.

Now, there's one big problem I need to talk about. So far, to encrypt a long message, you just divided it into blocks of 16 bytes (and perhaps you padded the last block). This naive way is called the *electronic codebook* (ECB) mode of operation. As you learned, encryption is deterministic, and so encrypting the same block of plaintext twice leads to the same ciphertext. This means that by encrypting each block individually, the resulting ciphertext might have repeating patterns.

This might seem fine, but allowing these repetitions lead to many problems. The most obvious one is that they leak information about the plaintext. The most famous illustration of this is the *ECB penguin*, pictured in figure 4.7.

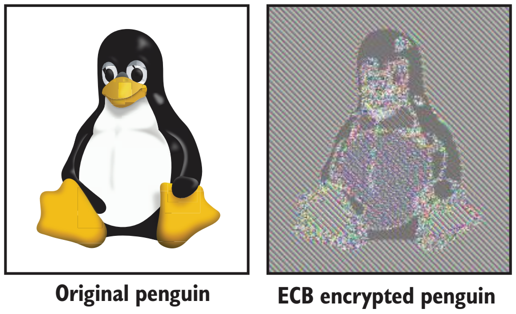


Figure 4.7 The famous ECB penguin is an encryption of an image of a penguin using the electronic codebook (ECB) mode of operation. As ECB does not hide repeating patterns, one can guess just by looking at the ciphertext what was originally encrypted. (Image taken from Wikipedia.)

To encrypt more than 128 bits of plaintext safely, better modes of operation exist that "randomize" the encryption. One of the most popular modes of operation for AES is *cipher block chaining* (CBC). CBC works for any deterministic block cipher (not just AES) by taking an additional value called an *initialization vector* (IV) to randomize the encryption. Because of this, the IV is the length of the block size (16 bytes for AES) and must be random and unpredictable.

To encrypt with the CBC mode of operation, start by generating a random IV of 16 bytes (chapter 8 tells you how to do this), then XOR the generated IV with the first 16 bytes of plaintext before encrypting those. This effectively randomizes the encryption. Indeed, if the same plaintext is encrypted twice but with different IVs, the mode of operation renders two different ciphertexts.

If there is more plaintext to encrypt, use the previous ciphertext (like we used the IV previously) to XOR it with the next block of plaintext before encrypting it. This randomizes the next block of encryption as well. Remember, the encryption of something is unpredictable and should be as good as the randomness we used to create our real IV. Figure 4.8 illustrates CBC encryption.

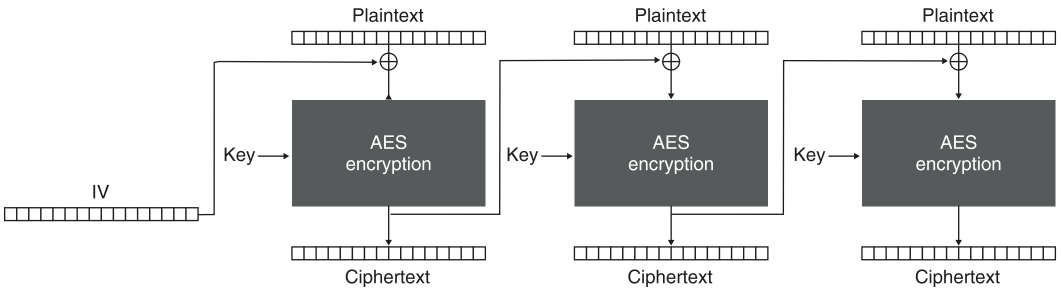


Figure 4.8 The CBC mode of operation with AES. To encrypt, we use a random initialization vector (IV) in addition to padded plaintext (split in multiple blocks of 16 bytes).

To decrypt with the CBC mode of operation, reverse the operations. As the IV is needed, it must be transmitted in clear text along with the ciphertext. Because the IV is supposed to be random, no information is leaked by observing the value. I illustrate CBC decryption in figure 4.9.

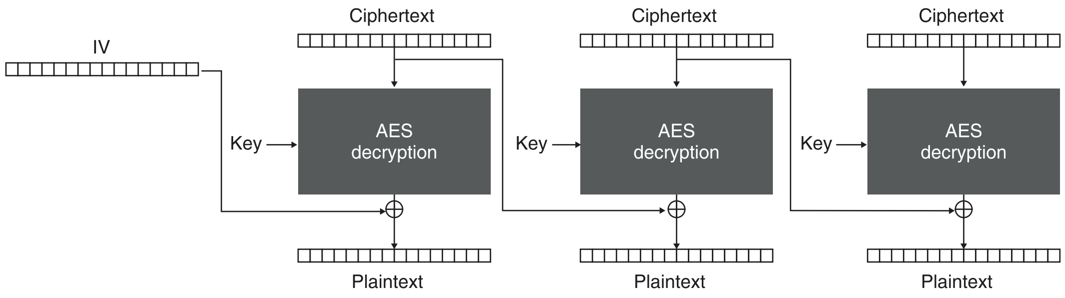


Figure 4.9 The CBC mode of operation with AES. To decrypt, the associated initialization vector (IV) is required.

Additional parameters like IVs are prevalent in cryptography. Yet, these are often poorly understood and are a great source of vulnerabilities. With the CBC mode of operation, an IV needs to be *unique* (it cannot repeat) as well as *unpredictable* (it really needs to be random). These requirements can fail for a number of reasons. Because developers are often confused by IVs, some cryptographic libraries have removed the possibility to specify an IV when encrypting with CBC and automatically generate one randomly.

WARNING When an IV repeats or is predictable, the encryption becomes deterministic again, and a number of clever attacks become possible. This was the case with the famous BEAST attack (Browser Exploit Against SSL/TLS) on the TLS protocol. Note also that other algorithms might have different requirements for IVs. This is why it is always important to read the manual. Dangerous details lie in fine print.

Note that a mode of operation and a padding are still not enough to make a cipher usable. You're about to see why in the next section.

4.4 A lack of authenticity, hence AES-CBC-HMAC

So far, we have failed to address one fundamental flaw: the ciphertext as well as the IV in the case of CBC can still be modified by an attacker. Indeed, there's no integrity mechanism to prevent that! Changes in the ciphertext or IV might have unexpected changes in the decryption. For example, in AES-CBC (AES used with the CBC mode of operation), an attacker can flip specific bits of plaintext by flipping bits in its IV and ciphertext. I illustrate this attack in figure 4.10.

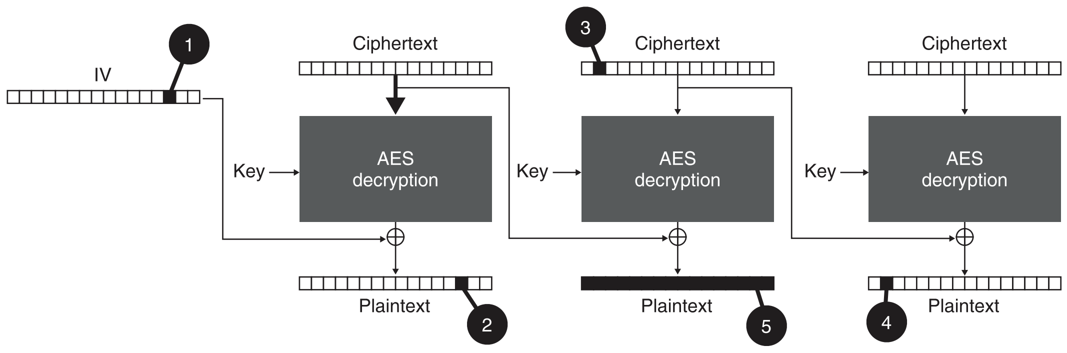


Figure 4.10 An attacker that intercepts an AES-CBC ciphertext can do the following: (1) Because the IV is public, flipping a bit (from 1 to 0, for example) of the IV also (2) flips a bit of the first block of plaintext. (3) Modifications of bits can happen on the ciphertext blocks as well. (4) Such changes impact the following block of decrypted plaintext. (5) Note that tampering with the ciphertext blocks has the direct effect of scrambling the decryption of that block.

Consequently, a cipher or a mode of operation must not be used as-is. They lack some sort of integrity protection to ensure that a ciphertext and its associated parameters (here the IV) cannot be modified without triggering some alarms.

To prevent modifications on the ciphertext, we can use the *message authentication codes* (MACs) that we saw in chapter 3. For AES-CBC, we usually use HMAC (for *hash-based MAC*) in combination with the SHA-256 hash function to provide integrity. We then apply the MAC after padding the plaintext and encrypting it over both the ciphertext and the IV; otherwise, an attacker can still modify the IV without being caught.

WARNING This construction is called *Encrypt-then-MAC*. The alternatives (like *MAC-then-Encrypt*) can sometimes lead to clever attacks (like the famous Vaudenay padding oracle attack) and are thus avoided in practice.

The created authentication tag can be transmitted along with the IV and the ciphertext. Usually, all are concatenated together as figure 4.11 illustrates. In addition, it is best practice to use different keys for AES-CBC and HMAC.

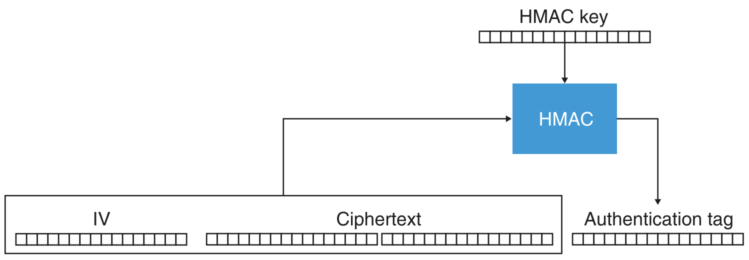


Figure 4.11 The AES-CBC-HMAC construction produces three arguments that are usually concatenated in the following order: the public IV, the ciphertext, and the authentication tag.

Prior to decryption, the tag needs to be verified (in constant time as you saw in chapter 3). The combination of all of these algorithms is referred to as *AES-CBC-HMAC* and was one of the most widely used authenticated encryption modes until we started to adopt more modern all-in-one constructions.

WARNING AES-CBC-HMAC is not the most developer-friendly construction. It is often poorly implemented and has some dangerous pitfalls when not used correctly (for example, the IV of each encryption *must* be unpredictable). I have spent a few pages introducing this algorithm as it is still widely used and still works, but I recommend against using it in favor of the more current constructions I introduce next.

4.5 All-in-one constructions: Authenticated encryption

The history of encryption is not pretty. Not only has it been poorly understood that encryption without authentication is dangerous, but misapplying authentication has also been a systemic mistake made by developers. For this reason, a lot of research has emerged seeking to standardize all-in-one constructions that simplify the use of encryption for developers. In the rest of this section, I introduce this new concept as well as two widely adopted standards: AES-GCM and ChaCha20-Poly1305.

4.5.1 What's authenticated encryption with associated data (AEAD)?

The most current way of encrypting data is to use an all-in-one construction called *authenticated encryption with associated data* (AEAD). The construction is extremely close to what AES-CBC-HMAC provides as it also offers confidentiality of your plaintexts while detecting any modifications that could have occurred on the ciphertexts. What's more, it provides a way to authenticate *associated data*.

The associated data argument is optional and can be empty or it can also contain metadata that is relevant to the encryption and decryption of the plaintext. This data will not be encrypted and is either implied or transmitted along with the ciphertext. In addition, the ciphertext's size is larger than the plaintext because it now contains an additional authentication tag (usually appended to the end of the ciphertext).

To decrypt the ciphertext, we are required to use the same implied or transmitted associated data. The result is either an error, indicating that the ciphertext was modified in transit, or the original plaintext. I illustrate this new primitive in figure 4.12.

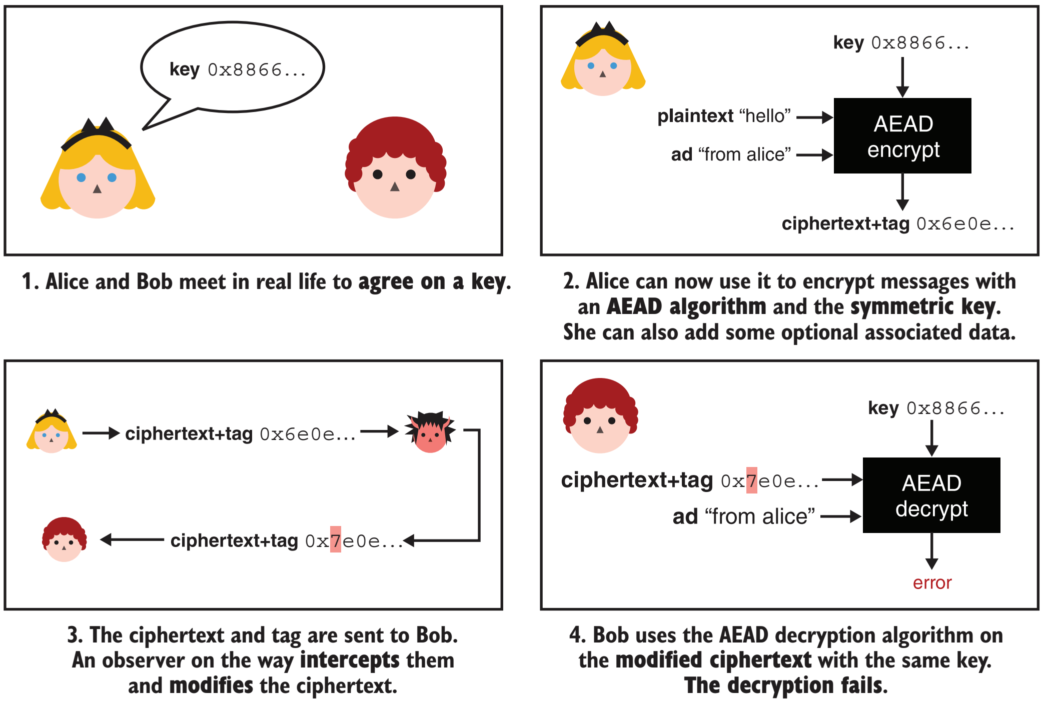

key 0x8866...
1. Alice and Bob meet in real life to agree on a key.
key 0x8866...
plaintext "hello"
ad "from alice"
AEAD encrypt
ciphertext+tag 0x6e0e...
2. Alice can now use it to encrypt messages with an AEAD algorithm and the symmetric key. She can also add some optional associated data.
ciphertext+tag 0x6e0e...
ciphertext+tag 0x7e0e...
3. The ciphertext and tag are sent to Bob. An observer on the way intercepts them and modifies the ciphertext.
key 0x8866...
ciphertext+tag 0x7e0e...
ad "from alice"
AEAD decrypt
error
4. Bob uses the AEAD decryption algorithm on the modified ciphertext with the same key. The decryption fails.

Figure 4.12 Both Alice and Bob meet in person to agree on a shared key. Alice can then use an AEAD encryption algorithm with the key to encrypt her messages to Bob. She can optionally authenticate some associated data (ad); for example, the sender of the message. After receiving the ciphertext and the authentication tag, Bob can decrypt it using the same key and associated data. If the associated data is incorrect or the ciphertext was modified in transit, the decryption fails.

Let's see how to use a *cryptographic library* to encrypt and decrypt with an authenticated encryption primitive. For this, we'll use the JavaScript programming language and the Web Crypto API (an official interface supported by most browsers that provides low-level cryptographic functions) as the following listing shows.

Listing 4.1 Authenticated encryption with AES-GCM in JavaScript

```
let config = {
    name: 'AES-GCM',
    length: 128          <- Generates a 128-bit key for 128 bits of security
};
let keyUsages = ['encrypt', 'decrypt'];
let key = await crypto.subtle.generateKey(config, false, keyUsages);

let iv = new Uint8Array(12);
await crypto.getRandomValues(iv);          <- Generates a 12-byte IV randomly

let te = new TextEncoder();
let ad = te.encode("some associated data");          <- Uses some associated data to encrypt our plaintext. Decryption must use the same IV and associated data.
let plaintext = te.encode("hello world");

let param = {
    name: 'AES-GCM',
    iv: iv,
    additionalData: ad
};
let ciphertext = await crypto.subtle.encrypt(param, key, plaintext);

let result = await window.crypto.subtle.decrypt(
    param, key, ciphertext);          <- Decryption throws an exception if the IV, ciphertext, or associated data are tampered with.
new TextDecoder("utf-8").decode(result);
```

Note that Web Crypto API is a low-level API, and as such, does not help the developer to avoid mistakes. For example, it lets us specify an IV, which is a dangerous pattern. In this listing, I used AES-GCM, which is the most widely used AEAD. Next, let's talk more about this AES-GCM.

4.5.2 The AES-GCM AEAD

The most widely used AEAD is AES with the *Galois/Counter Mode* (also abbreviated AES-GCM). It was designed for high performance by taking advantage of hardware support for AES and by using a MAC (GMAC) that can be implemented efficiently.

AES-GCM has been included in NIST's Special Publication (SP 800-38D) since 2007, and it is the main cipher used in cryptographic protocols, including several versions of the TLS protocol that is used to secure connections to websites on the internet. Effectively, we can say that AES-GCM encrypts the web.

AES-GCM combines the Counter (CTR) mode of operation with the GMAC message authentication code. First, let's see how CTR mode works with AES. Figure 4.13 shows how AES is used with CTR mode.

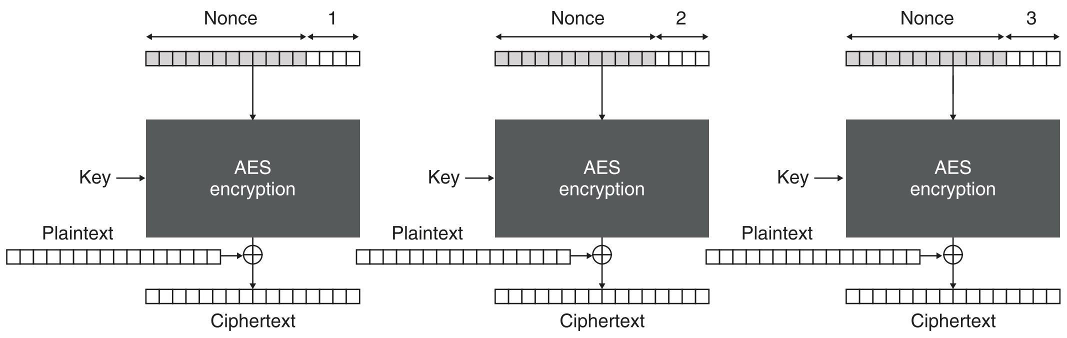


Figure 4.13 The AES-CTR algorithm combining the AES cipher with the Counter mode of operation (CTR mode). A unique nonce is concatenated with a counter and encrypted to produce a keystream. The keystream is then XORed with the actual bytes of the plaintext.

AES-CTR uses AES to encrypt a nonce concatenated with a number (starting at 1) instead of the plaintext. This additional argument, "a nonce for number once," serves the same purpose as an IV: it allows the mode of operation to randomize the AES encryption. The requirements are a bit different from the IV of CBC mode, however. A nonce needs to be unique but *not* unpredictable. Once this 16-byte block is encrypted, the result is called a *keystream*, and it is XORed with the actual plaintext to produce the encryption.

NOTE Like IVs, nonces are a common term in cryptography, and they are found in different cryptographic primitives. Nonces can have different requirements, although the name often indicates that it should not repeat. But, as usual, what matters is what the manual says, not what the name of the argument implies. Indeed, the nonce of AES-GCM is sometimes referred to as an IV.

The nonce in AES-CTR is 96 bits (12 bytes) and takes most of the 16 bytes to be encrypted. The 32 bits (4 bytes) left serves as a counter, starting from 1 and incremented for each block encryption until it reaches its maximum value at $2^{4\times8} - 1 =$ 4,294,967,295. This means that, at most, 4,294,967,295 blocks of 128 bits can be encrypted with the same nonce (so less than 69 GBs).

If the same nonce is used twice, the same keystream is created. By XORing the two ciphertexts together, the keystream is canceled and one can recover the XOR of the two plaintexts. This can be devastating, especially if you have some information about the content of one of the two plaintexts.

Figure 4.14 shows an interesting aspect of CTR mode: no padding is required. We say that it turns a block cipher (AES) into a stream cipher. It encrypts the plaintext byte by byte.

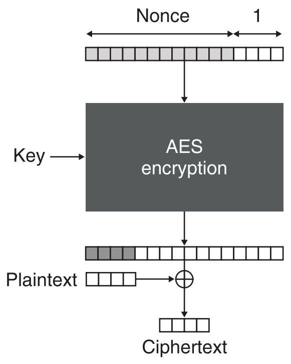


Figure 4.14 If the keystream of AES-CTR is longer than the plaintext, it is truncated to the length of the plaintext prior to XORing it with the plaintext. This permits AES-CTR to work without padding.

Stream ciphers

Stream ciphers are another category of ciphers. They are different than block ciphers because we can use them directly to encrypt a ciphertext by XORing it with a keystream. No need for padding or a mode of operation, allowing the ciphertext to be of the same length as the plaintext.

In practice, there isn't much difference between these two categories of ciphers because block ciphers can easily be transformed into stream ciphers via the CTR mode of operation. But, in theory, block ciphers have the advantage as they can be useful when constructing other categories of primitives (similar to what you saw in chapter 2 with hash functions).

This is also a good moment to note that, by default, encryption doesn't (or badly) hides the length of what you are encrypting. Because of this, the use of compression before encryption can lead to attacks if an attacker can influence parts of what is being encrypted.

The second part of AES-GCM is *GMAC*. It is a MAC constructed from a keyed hash (called *GHASH*). In technical terms, GHASH is an almost XORed universal hash (AXU), which is also called a *difference unpredictable function* (DUF). The requirement of such a function is weaker than a hash. For example, an AXU does not need to be collision resistant. Thanks to this, GHASH can be significantly faster. Figure 4.15 illustrates the GHASH algorithm.

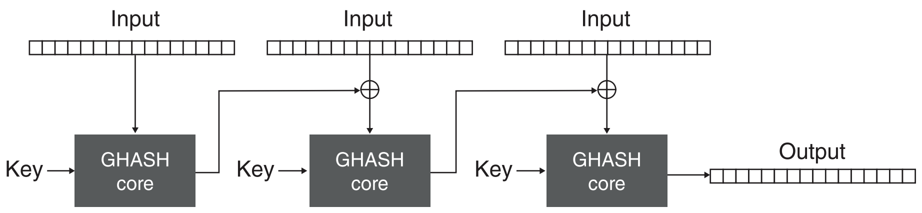


Figure 4.15 GHASH takes a key and absorbs the input block by block in a manner resembling CBC mode. It produces a digest of 16 bytes.

To hash something with GHASH, we break the input into blocks of 16 bytes and then hash them in a way similar to CBC mode. As this hash takes a key as input, it can theoretically be used as a MAC, but only once (otherwise, the algorithm breaks)—it's a *one-time MAC*. As this is not ideal for us, we use a technique (due to Wegman-Carter) to transform GHASH into a *many-time MAC*. I illustrate this in figure 4.16.

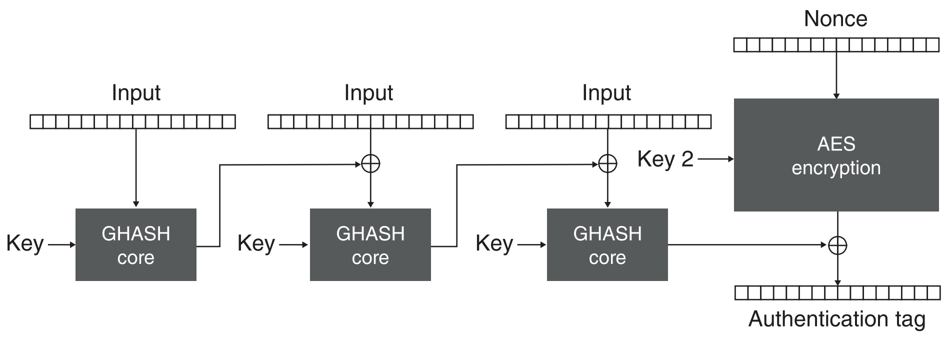


Figure 4.16 GMAC uses GHASH with a key to hash the input, then encrypts it with a different key and AES-CTR to produce an authentication tag.

GMAC is effectively the encryption of the GHASH output with AES-CTR (and a different key). Again, the nonce must be unique; otherwise, clever attackers can recover the authentication key used by GHASH, which would be catastrophic and would allow easy forgery of authentication tags.

Finally, AES-GCM can be seen as an intertwined combination of CTR mode and GMAC, similar to the Encrypt-then-MAC construction we previously discussed. I illustrate the whole algorithm in figure 4.17.

The counter starts at 1 for encryption, leaving the 0 counter for encrypting the authentication tag created by GHASH. GHASH, in turn, takes an independent key H, which is the encryption of the all-zero block with a key K. This way one does not need to carry two different keys as the key K suffices to derive the other one.

As I said previously, the 12-byte nonce of AES-GCM needs to be unique and, thus, to never repeat. Notice that it doesn't need to be random. Consequently, some people like to use it as a *counter*, starting it at 1 and incrementing it for each encryption. In this case, one must use a cryptographic library that lets the user choose the nonce. This allows one to encrypt $2^{12\times8} - 1$ messages before reaching the maximum value of the nonce. Suffice it to say, this is an impossible number of messages to reach in practice.

On the other hand, having a counter means that one needs to keep *state*. If a machine crashes at the wrong time, it is possible that nonce reuse could happen. For this reason, it is sometimes preferred to have a *random nonce*. Actually, some libraries will not let developers choose the nonce and will generate those at random. Doing

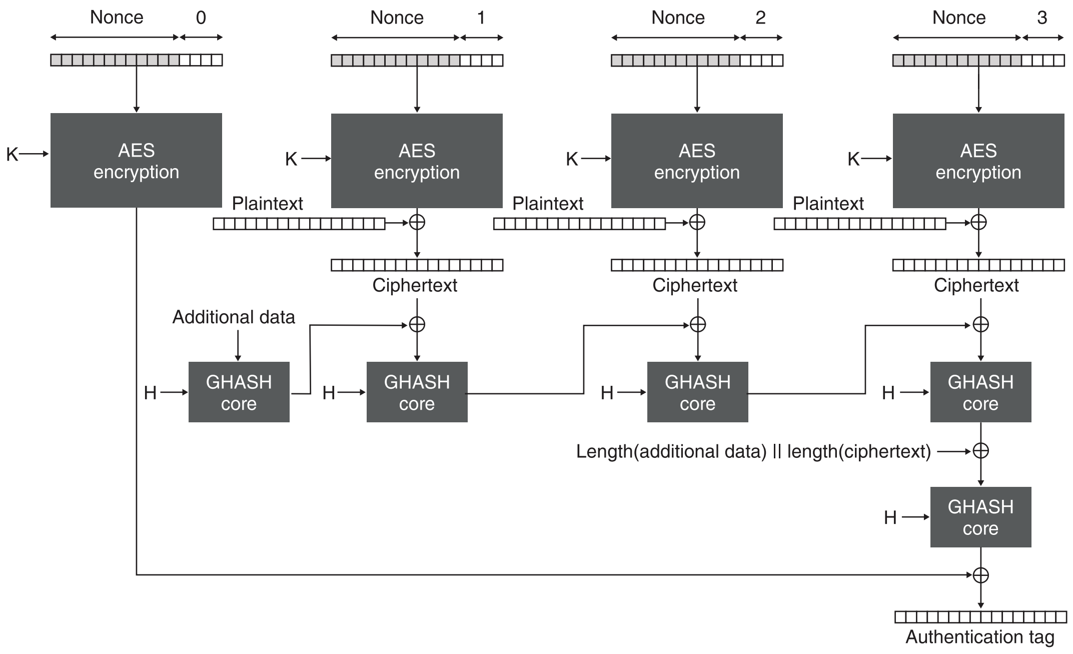


Figure 4.17 AES-GCM works by using AES-CTR with a symmetric key *K* to encrypt the plaintext and by using GMAC to authenticate the associated data and the ciphertext using an authentication key *H*.

this avoids repetition with probabilities so high that this shouldn't happen in practice. Yet, the more messages that are encrypted, the more nonces are used and the higher the chances of getting a collision. Because of the birthday bound we talked about in chapter 2, it is recommended not to encrypt more than $2^{92/3} \approx 2^{30}$ messages with the same key when generating nonces randomly.

Beyond birthday-bound security

2^{30} messages is quite a large number of messages. It might never be reached in many scenarios, but real-world cryptography often pushes the limit of what is considered reasonable. Some long-lived systems need to encrypt many, many messages per second, eventually reaching these limits. Visa, for example, processes 150 million transactions per day. If it needs to encrypt those with a unique key, it would reach the limit of 2^{30} messages in only a week. In these extreme cases, *rekeying* (changing the key used to encrypt) can be a solution. There also exists a research field called *beyond birthday-bound security* that aims to improve the maximum number of messages that can be encrypted with the same key.

4.5.3 ChaCha20-Poly1305

The second AEAD I will talk about is *ChaCha20-Poly1305*. It is the combination of two algorithms: the ChaCha20 stream cipher and the Poly1305 MAC. Both designed separately by Daniel J. Bernstein to be fast when used in software, contrary to AES, which is slow when hardware support is unavailable. In 2013, Google standardized the ChaCha20-Poly1305 AEAD in order to make use of it in Android mobile phones relying on low-end processors. Nowadays, it is widely adopted by internet protocols like OpenSSH, TLS, and Noise.

ChaCha20 is a modification of the Salsa20 stream cipher, which was originally designed by Daniel J. Bernstein around 2005. It was one of the nominated algorithms in the ESTREAM competition (https://www.ecrypt.eu.org/stream/). Like all stream ciphers, the algorithm produces a *keystream*, a series of random bytes of the length of the plaintext. It is then XORed with the plaintext to create the ciphertext. To decrypt, the same algorithm is used to produce the same keystream, which is XORed with the ciphertext to give back the plaintext. I illustrate both flows in figure 4.18.

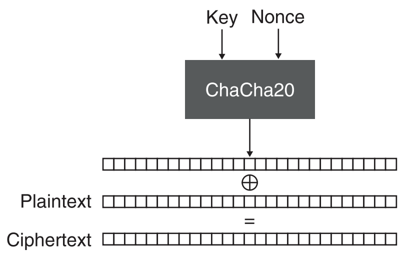


Figure 4.18 ChaCha20 works by taking a symmetric key and a unique nonce. It then generates a keystream that is XORed with the plaintext (or ciphertext) to produce the ciphertext (or plaintext). The encryption is length-preserving as the ciphertext and the plaintext are of the same length.

Under the hood, ChaCha20 generates a keystream by repeatedly calling a *block function* to produce many 64-byte blocks of keystream. The block function takes

- A 256-bit (32-byte) key like AES-256
- A 92-bit (12-byte) nonce like AES-GCM
- A 32-bit (4-byte) counter like AES-GCM

The process to encrypt is the same as with AES-CTR. (I illustrate this flow in figure 4.19.)

1. Run the block function, incrementing the counter every time, until enough keystream is produced
2. Truncate the keystream to the length of the plaintext
3. XOR the keystream with the plaintext

Due to the upper bound on the counter, you can use ChaCha20 to encrypt as many messages as with AES-GCM (as it is parameterized by a similar nonce). Because the output created by this block function is much larger, the size of a message that you can encrypt is also impacted. You can encrypt a message of size $2^{32} \times 64$ bytes ≈ 274 GB. If a nonce is reused to encrypt a plaintext, similar issues to AES-GCM arise. An observer

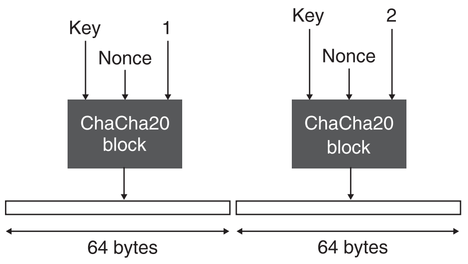


Figure 4.19 ChaCha20's keystream is created by calling an internal block function until enough bytes are produced. One block function call creates 64 bytes of random keystream.

can obtain the XOR of the two plaintexts by XORing the two ciphertexts and can also recover the authentication key for the nonce. These are serious issues that can lead to an attacker being able to forge messages!

The size of nonces and counters

The size of the nonces and the counters are (actually) not always the same everywhere (both for AES-GCM and ChaCha20-Poly1305), but they are the recommended values from the adopted standards. Still, some cryptographic libraries accept different sizes of nonce, and some applications increase the size of the counter (or the nonce) in order to allow encryption of larger messages (or more messages). Increasing the size of one component necessarily decreases the size of the other.

To prevent this, while allowing a large number of messages to be encrypted under a single key, other standards like XChaCha20-Poly1305 are available. These standards increase the size of the nonce while keeping the rest intact, which is important in cases where the nonce needs to be generated randomly instead of being a counter tracked in the system.

Inside the ChaCha20 block function, a state is formed. Figure 4.20 illustrates this state.

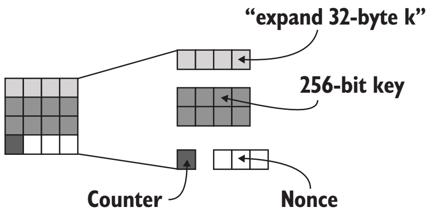


Figure 4.20 The state of the ChaCha20 block function. It is formed by 16 words (represented as squares) of 32 bytes each. The first line stores a constant, the second and third lines store the 32-byte symmetric key, the following word stores a 4-byte counter, and the last 3 words store the 12-byte nonce.

This state is then transformed into 64 bytes of keystream by iterating a round function 20 times (hence the 20 in the name of the algorithm). This is similar to what was done with AES and its round function. The round function is itself calling a *Quarter Round* (QR) *function* 4 times per round, acting on different words of the internal state each time, depending if the round number is odd or even. Figure 4.21 shows this process.

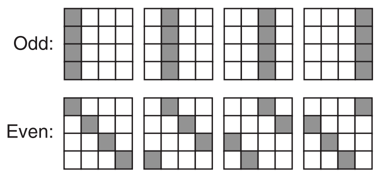


Figure 4.21 A round in ChaCha20 affects all the words contained in a state. As the Quarter Round (QR) function only takes 4 arguments, it must be called at least 4 times on different words (grayed in the diagram) to modify all 16 words of the state.

The QR function takes four different arguments and updates them using only Add, Rotate, and XOR operations. We say that it is an *ARX* stream cipher. This makes ChaCha20 extremely easy to implement and fast in software.

Poly1305 is a MAC created via the Wegman-Carter technique, much like the GMAC we previously talked about. Figure 4.22 illustrates this cryptographic MAC.

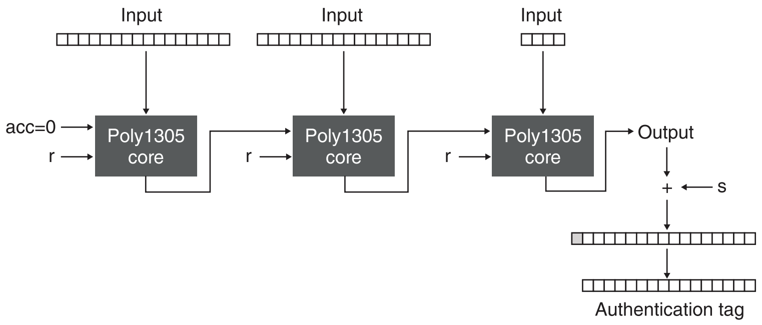


Figure 4.22 Poly1305's core function absorbs an input one block at a time by taking an additional accumulator set to 0 initially and an authentication key *r*. The output is fed as an accumulator to the next call of the core function. Eventually the output is added to a random value *s* to become the authentication tag.

In the figure, *r* can be seen as the authentication key of the scheme, like the authentication key *H* of GMAC. And *s* makes the MAC secure for multiple uses by encrypting the result, thus it must be unique for each usage.

The *Poly1305 core function* mixes the key with the accumulator (set to 0 in the beginning) and the message to authenticate. The operations are simple multiplications modulo a constant *P*.

> **NOTE** Obviously, a lot of details are missing from our description. I seldom mention how to encode data or how some arguments should be padded before being acted on. These are all implementation specificities that do not matter for us as we are trying to get an intuition of how these things work.

Eventually, we can use ChaCha20 and a counter set to 0 to generate a keystream and derive the 16-byte *r* and 16-byte *s* values we need for Poly1305. I illustrate the resulting AEAD cipher in figure 4.23.

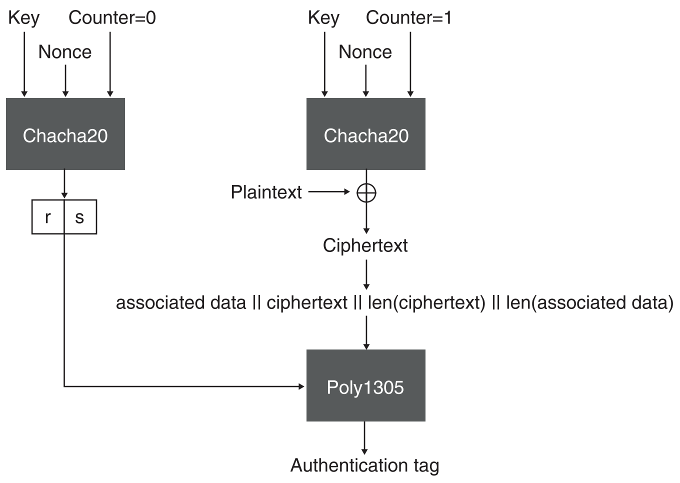


Figure 4.23 ChaCha20-Poly1305 works by using ChaCha20 to encrypt the plaintext and to derive the keys required by the Poly1305 MAC. Poly1305 is then used to authenticate the ciphertext as well as the associated data.

The normal ChaCha20 algorithm is first used to derive the authentication secrets *r* and *s* needed by Poly1305. The counter is then incremented, and ChaCha20 is used to encrypt the plaintext. After that, the associated data and the ciphertext (and their respective lengths) are passed to Poly1305 to create an authentication tag.

To decrypt, the exact same process is applied. ChaCha20 first verifies the authentication of the ciphertext and the associated data via the tag received. It then decrypts the ciphertext.

4.6 Other kinds of symmetric encryption

Let's pause for a moment and review the symmetric encryption algorithms you have learned so far:

- *Non-authenticated encryption*—AES with a mode of operation but without a MAC. This is insecure in practice as ciphertexts can be tampered with.
- *Authenticated encryption*—AES-GCM and ChaCha20-Poly1305 are the two most widely adopted ciphers.

The chapter could end here and it would be fine. Yet, real-world cryptography is not always about the agreed standards; it is also about *constraints* in size, in speed, in format, and so on. To that end, let me give you a brief overview of other types of symmetric encryption that can be useful when AES-GCM and ChaCha20-Poly1305 won't fit.

4.6.1 Key wrapping

One of the problems of nonce-based AEADs is that they all require a nonce, which takes additional space. Notice that when encrypting a key, you might not necessarily need randomization because what is encrypted is already random and will not repeat with high probabilities (or if it does repeat, it is not a big deal). One well-known standard for key wrapping is NIST's Special Publication 800-38F: "Recommendation for

Block Cipher Modes of Operation: Methods for Key Wrapping." These key wrapping algorithms do not take an additional nonce or IV and randomize their encryption based on what they are encrypting. Thanks to this, they do not have to store an additional nonce or IV next to the ciphertexts.

4.6.2 Nonce misuse-resistant authenticated encryption

In 2006, Phillip Rogaway published a new key wrapping algorithm called *synthetic initialization vector* (SIV). As part of the proposal, Rogaway notes that SIV is not only useful to encrypt keys, but also as a general AEAD scheme that is more tolerant to nonce repetitions. As you learned in this chapter, a repeating nonce in AES-GCM or ChaCha20-Poly1305 can have catastrophic consequences. It not only reveals the XOR of the two plaintexts, but it also allows an attacker to recover an authentication key and to forge valid encryption of messages.

The point of a nonce misuse-resistant algorithm is that encrypting two plaintexts with the same nonce only reveals if the two plaintexts are equal or not, and that's it. It's not great, but it's obviously not as bad as leaking an authentication key. The scheme has gathered a lot of interest and has since been standardized in RFC 8452: "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption." The trick behind SIV is that the nonce used in the AEAD to encrypt is generated from the plaintext itself, which makes it highly unlikely that two different plaintexts would end up being encrypted under the same nonce.

4.6.3 Disk encryption

Encrypting the storage of a laptop or a mobile phone has some hefty constraints: it has to be fast (otherwise the user will notice) and you can only do it in place (saving space is important for a large number of devices). Because the encryption can't expand, AEADs that need a nonce and an authentication tag are not a good fit. Instead, unauthenticated encryption is used.

To protect against *bitflip attacks*, large blocks (think thousands of bytes) of data are encrypted in a way that a single bitflip would scramble the decryption of the whole block. This way, an attack has more of a chance of crashing the device than accomplishing its goal. These constructions are called *wide-block ciphers*, although this approach has also been dubbed *poor man's authentication*.

Linux systems and some Android devices have adopted this approach using Adiantum, a wide-block construction wrapping the ChaCha cipher and standardized by Google in 2019. Still, most devices use non-ideal solutions: both Microsoft and Apple make use of AES-XTS, which is unauthenticated and is not a wide-block cipher.

4.6.4 Database encryption

Encrypting data in a database is tricky. As the whole point is to prevent database breaches from leaking data, the key used to encrypt and decrypt the data must be stored away from the database server. Because clients don't have the data themselves, they are severely limited in the way they can query the data.

The simplest solution is called *transparent data encryption* (TDE) and simply encrypts selected columns. This works well in some scenarios, although one needs to be careful to authenticate associated data identifying the row and the column being encrypted; otherwise, encrypted content can be swapped. Still, one cannot search through encrypted data and so queries have to use the unencrypted columns.

Searchable encryption is the field of research that aims at solving this problem. A lot of different schemes have been proposed, but it seems like there is no silver bullet. Different schemes propose different levels of "searchability" as well as different degradations in security. Blind indexing, for example, simply allows you to search for exact matches, while order-preserving and order-revealing encryptions allow you to order results. The bottom line is, the security of these solutions are to be looked at carefully as they truly are tradeoffs.

Summary

- Encryption (or symmetric encryption) is a cryptographic primitive that can be used to protect the confidentiality of data. The security relies on a symmetric key that needs to remain secret.
- Symmetric encryption needs to be authenticated (after which we call it authenticated encrption) to be secure, as otherwise, ciphertexts can be tampered with.
- Authenticated encryption can be constructed from a symmetric encryption algorithm by using a message authentication code. But best practice is to use an authenticated encryption with associated data (AEAD) algorithm as they are all-in-one constructions that are harder to misuse.
- Two parties can use authenticated encryption to hide their communications, as long as they both have knowledge of the same symmetric key.
- AES-GCM and ChaCha20-Poly1305 are the two most widely adopted AEADs. Most applications nowadays use either one of these.
- Reusing nonces breaks the authentication of AES-GCM and ChaCha20-Poly1305. Schemes like AES-GCM-SIV are nonce misuse resistant, while encryption of keys can avoid that problem as nonces are not necessary.
- Real-world cryptography is about constraints, and AEADs cannot always fit every scenario. This is the case for database or disk encryption, for example, that require the development of new constructions.